Reprint from
The *Mathematica* Conference
June, 1992   Boston, MA

881.21.10.1992

Chapter 6 Graphics Programming

This chapter provides an introduction to the use of graphics with the rest of the *Mathematica* system. First we quickly review programming in *Mathematica*. Then we consider building functions which draw on the power of the *Mathematica* system. This is followed by a discussion of animation. The chapter concludes with some examples of graphics programming.

One of the most powerful aspects of *Mathematica* graphics is the availability of the rest of the system. This provides many capabilities and functions which are not normally present in a graphics application. One can use the built-in *Mathematica* functions which implement a wide range of numerical techniques such as root-finding, function minimization, numerically solving differential equations or quadrature, and then graph the results. Further still one can write code to construct primitives which draw a special type of picture.

For more information on *Mathematica* programming consider the book *Mathematica: A System for Doing Mathematics by Computer*, Second Edition, by Stephen Wolfram, *Mathematica* conference notes which are available from Wolfram Research or books on the subject such as *Programming in Mathematica* by Roman Maeder or *Mathematica: A Practical Approach* by Nancy Blachman.

6.1 Introduction to *Mathematica* Programming

Mathematica Patterns and Rules

One of the unique things about *Mathematica* programming is the ease and economy with which rules are given to describe how a function should behave. If you don't like the way it works, add a rule to make it work just as you desire.

f`[1]`	matches exactly *f*`[1]`
f`[_]`	matches *f*`[` *thing* `]`
f`[x_]`	matches *f*`[` *thing* `]` and gives *thing* a name

Some simple *Mathematica* patterns.

This gives a rule for `f[1]`.

```
In[1]:= f[1] := foo
```

This evaluates since a rule matches the input.

```
In[2]:= f[1]
Out[2]= foo
```

This does not evaluate since there is no rule to match it.

```
In[3]:= f[2]
Out[3]= f[2]
```

This will match `f[` *thing* `]`.

```
In[4]:= f[ _] := bar
```

Now this matches a pattern and the result is returned.

```
In[5]:= f[2]
Out[5]= bar
```

This does not match since a pattern has only been for `f` of one argument.

```
In[6]:= f[ 2,3]
Out[6]= f[2, 3]
```

This will clear all the rules of `f`.

```
In[7]:= Clear[ f]
```

This gives *thing* a name, namely `x`.

```
In[8]:= f[ x_] := x^3 - x^2
```

This matches.

```
In[9]:= f[ 5]
Out[9]= 100
```

And this matches the pattern as well.

```
In[10]:= f[ (a + b)]
                2          3
Out[10]= -(a + b)  + (a + b)
```

More complex patterns can be given to match multiple arguments, to refer to particular types of inputs and to accept conditions. The book *Mathematica: A System for Doing Mathematics by Computer*, Second Edition, describes patterns is considerably more detail.

Delayed Evaluation

Often in these notes we see `=` and `:=` which are the assignment functions. The first evaluates the right-hand side before the assignment is made. The second, `:=` which is delayed assignment, does not evaluate the right-hand side until the left-hand side is evaluated.

`=`	evaluate and then assign
`:=`	assign but don't evaluate

This evaluates before assignment.

```
In[11]:= f1 = Random[]

Out[11]= 0.0560708
```

These are all the same since `f1` has been set to the value of the random number.

```
In[12]:= Table[ f1, {5}]

Out[12]= {0.0560708, 0.0560708, 0.0560708, 0.0560708, 0.0560708}
```

This makes a delayed assignment.

```
In[13]:= f2 := Random[]
```

We now see a list of different and random numbers. This is because the evaluation was delayed.

```
In[14]:= Table[ f2, {5}]

Out[14]= {0.6303, 0.359894, 0.871377, 0.858645, 0.584579}

In[15]:= ??f1

Global`f1

f1 = 0.05607079483486153
```

The settings of the `??` clearly show the difference.

```
In[15]:= ??f2

    Global`f2

f2 := Random[]
```

Delayed assignment avoids evaluating the left-hand side. There are a number of other functions which do not evaluate their arguments. An example is `Plot`, which possesses the attribute `HoldAll` delaying the evaluation of its arguments. An attribute is a property of a symbol which affects its behavior. In the example here the attribute `HoldAll` controls expressions of which the symbol is the head. Attributes are discussed in further detail in The *Mathematica* Book.

`HoldAll`	a `HoldAll` function will delay argument evaluation
`Evaluate[` *expr*`]`	evaluate this even if in a `HoldAll` function
`Hold[` *expr* `]`	do not evaluate *expr*
`ReleaseHold[` *expr*`]`	remove `Hold` if present

Other ways to delay evaluation.

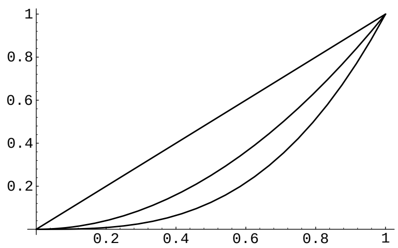
This shows that `Plot` delays evaluation of the arguments.

```
In[15]:= Attributes[ Plot]

Out[15]= {HoldAll, Protected}
```

This would not plot three functions without the `Evaluate`.

```
In[16]:= Plot[ Evaluate[ Table[ x^n, {n,3}]], {x,0,1}]
```



This does not evaluate due to `Hold`.

```
In[17]:= Hold[ 1 + 5]

Out[17]= Hold[1 + 5]
```

`ReleaseHold` removes the `Hold` and evaluation proceeds.

```
In[18]:= ReleaseHold[ %]

Out[18]= 6
```

Working with Lists

Mathematica lists are fundamental to the whole system. A good understanding of the power of list manipulations is a key to an understanding of *Mathematica* itself. There are many functions which work with lists and here we review some of the most basic.

`Table`	construct a list
`Map[` *f*`,` *list*`]`	travel down *list* applying *f* to each element
`Apply[` *f*`,` *list*`]`	make *f* the head of *list*
`Transpose[ {`*list1*`,` *list2*`}]`	transpose `{`*list1*`,` *list2*`}`

Some useful commands for working with lists.

Make a list.

```
In[19]:= list = Table[ i, {i,5}]

Out[19]= {1, 2, 3, 4, 5}
```

This travels down the list applying g to everything. It returns a list as the result.

```
In[20]:= Map[ g, list]

Out[20]= {g[1], g[2], g[3], g[4], g[5]}

In[21]:= Map[ Sqrt, list]

Out[21]= {1, Sqrt[2], Sqrt[3], 2, Sqrt[5]}
```

This exchanges the head of list with g.

```
In[22]:= Apply[ g, list]

Out[22]= g[1, 2, 3, 4, 5]
```

Of course the result may evaluate further. Using `Times` will multiply all the numbers together.

```
In[23]:= Apply[ Times, list]

Out[23]= 120

In[24]:= list1 = {a,b,c}

Out[24]= {a, b, c}

In[25]:= list2 = {1,2,3}

Out[25]= {1, 2, 3}
```

This is an extremely useful way to collect things together.

```
In[26]:= Transpose[ {list1, list2}]

Out[26]= {{a, 1}, {b, 2}, {c, 3}}
```

Pure Functions

A command like `Map` will march down a list and apply a function to each element of the list. The examples were all of things like `Sqrt` and were deliberately chosen to be functions which only took one argument which was used to build an expression. For example `Sqrt` would build things of the type `Sqrt[ 4]` which will evaluate further. But what do you do if have a function which takes two arguments or one which uses it's argument at a deeper level?

For an example let us say we want to go down a list and record whether the elements are greater or less than some number. We could make a function to do this.

This is my function.

```
In[27]:= newfun[ x_] := Greater[ x, 5]
```

This does the job.

```
In[28]:= Map[ newfun, {2,4,6,8}]

Out[28]= {False, False, True, True}
```

However this is not very satisfactory. I may want to compare my list with many different numbers and not just 5 as in this example. One solution is to use a pure function. This is a function which will put its argument at any level.

This is a pure function. It is labeled as a pure function by the & and the # shows where the argument will go.

```
In[29]:= Greater[ #, 5]&

Out[29]= #1 > 5 &
```

This applies the pure function. When this is evaluated the 10 will replace the #, called the slot, and this evaluates.

```
In[30]:= Greater[ #, 5]& [10]

Out[30]= True
```

This combines the pure function with Map.

```
In[31]:= Map[ Greater[ #, 5]&, {2,4,6,8}]

Out[31]= {False, False, True, True}
```

Blocks and Modules

If you have lines of code with a common purpose and wish to group them together this may be done with either `Block` or `Module`. These allow you to collect commands together and provides tools for local variables.

- `Block[ {`*locals*`},` *com1*`; ...` *com2*`]`
- `Module[ {`*locals*`},` *com1*`; ...` *com2*`]`

The result of `Block` is the last evaluation.

```
In[32]:= blockfun[  n_] :=
            Block[ { i, res},
              res = Table[ i, {i, n}] ;
              res = Partition[ res, 2] ;
              Transpose[ res]
            ]

In[33]:= blockfun[ 10]

Out[33]= {{1, 3, 5, 7, 9}, {2, 4, 6, 8, 10}}
```

`Module` is very similar to `Block`. The difference is the treatment of local variables.

```
In[34]:= modfun[  n_] :=
            Module[ { i, res},
              res = Table[ i, {i, n}] ;
              res = Partition[ res, 2] ;
              Transpose[ res]
            ]

In[35]:= modfun[ 12]

Out[35]= {{1, 3, 5, 7, 9, 11}, {2, 4, 6, 8, 10, 12}}
```

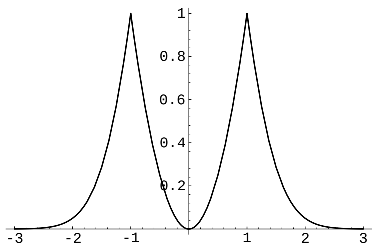
6.2 Function Plotting

Constructing a new function is very simple.

```
In[36]:= weirdf[x_] := If[ x > -1 && x < 1, x^2, Exp[ 1- x^2] ]
```

The function which `Plot` requires must evaluate to a number. Thus our weird function is quite adequate.

```
In[37]:= Plot[ weirdf[t], {t,-3,3}]
```



A further possibility is to use *Mathematica* numerical capabilities. For this the `NDSolve` function, which numerically solves differential equations, is very suitable.

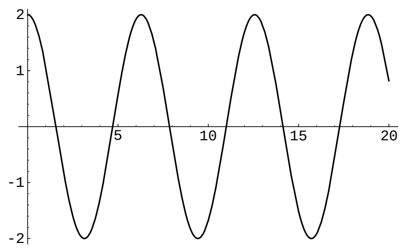
`NDSolve` returns an `InterpolatingFunction`. Of course we could have solved this problem analytically.

```
In[38]:= NDSolve[ {x''[t] == -x[t],
                   x[0] == 2,
                   x'[0] == 0}, x, {t,0,20}]

Out[38]= {{x -> InterpolatingFunction[{0., 20.}, <>]}}
```

This plots the result. `Evaluate` is necessary since `Plot` is `HoldAll`.

```
In[39]:= Plot[ Evaluate[ x[t] /. %], {t,0,20}]
```



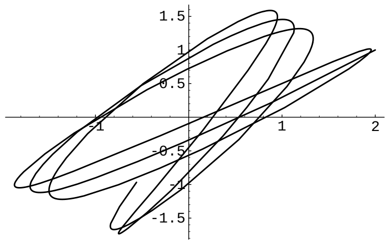
This example is more complex. It comprises of two coupled oscillators.

```
In[40]:= NDSolve[ {x''[t] == -x[t] - x[t] y[t]^2,
                   y''[t] == -y[t] -x[t] y[t]^2,
                   x[0] == 2, x'[0] == 0,
                   y[0] == 1, y'[0] == 0},
                   {x,y}, {t,0,20}]

Out[40]= {{x -> InterpolatingFunction[{0., 20.}, <>],
   y -> InterpolatingFunction[{0., 20.}, <>]}}
```

This shows the solution moving between x and y.

```
In[41]:= ParametricPlot[ Evaluate[ {x[t], y[t]} /. First[%]],
         {t,0,20}]
```



6.3 Animation

The animation of *Mathematica* graphics was introduced in an earlier chapter. The use of the animation package to carry out animations was demonstrated. Here we return to the subject and discuss it in the light of some of the ideas which have been covered in intervening chapters.

`<<Graphics`Animation``	load the animation package (it may be loaded)
`Animate[`*plot*`, {`*t*`, `*tmin*`, `*tmax*`}]`	execute the graphics command *plot* for a sequence of values of *t*, and animate the resulting sequence of frames
`ShowAnimation[{`g_1`, `g_2`, … }]`	produce an animation from a sequence of graphics objects

Commands defined in the package `Graphics`Animation``.

To demonstrate the effects of an animation in notes like these is quite difficult. One way to do this is to use the package `FlipBookAnimation`` which produces an array of images.

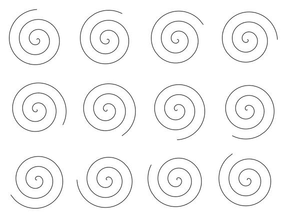
Do not load `FlipBookAnimation`` unless you wish to make a static sequence of pictures like these. If you wish an animated image your system is already configured to work.

```
In[42]:= <<FlipBookAnimation`
```

```
In[43]:= <<Graphics`Animation`
```

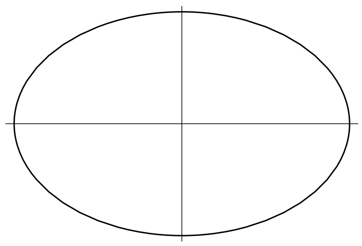
Here both `PlotRange` and `AspectRatio` are set. This is to enhance the effect of the animation.

```
In[44]:= Animate[
        ParametricPlot[ {t Sin[t+n], t Cos[t+n]}, {t,0,6Pi},
          AspectRatio -> Automatic,
          PlotRange -> {{-20, 20},{-20,20}}, Axes -> False],
        {n,0,2Pi-Pi/6, Pi/6}]
```



As shown here it is very often necessary to give explicit settings to `PlotRange`. This ensures that objects with the same coordinates do not move from frame to frame. As another example of an animation let's consider an ellipse.

```
In[45]:= plot = ParametricPlot[ { 1.5 Sin[x] , Cos[x]},
       {x,0,2Pi},
            AspectRatio -> Automatic, Ticks -> None]
```



This makes a list of primitives which will move a point around the edge of the ellipse. It will also join lines from the foci to this point. In a real application it may be interesting to decrease the step size.

```
In[46]:= prims =
      Table[{ PointSize[ 0.04],
              Point[{1.5 Sin[t], Cos[t]}],
              Thickness[ 0.01],
       Line[ {{-Sqrt[ 1.5^2 - 1.], 0},
                     {1.5 Sin[t], Cos[t]},
                     {Sqrt[ 1.5^2 - 1.], 0}}]},
       {t,0,2Pi-Pi/4,Pi/4}];
```

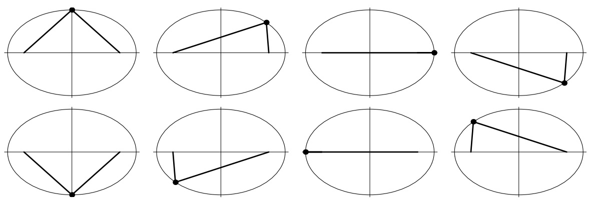
This produces a list of graphics objects, each of which is a different frame.

```
In[47]:= tab = Map[ (plot /. (Epilog -> {}) -> (Epilog -> #))&,
     prims]

Out[47]= {-Graphics-, -Graphics-, -Graphics-, -Graphics-,
  -Graphics-, -Graphics-, -Graphics-, -Graphics-}
```

Now we can show the result with the command `ShowAnimation`.

```
In[48]:= ShowAnimation[ tab]
```



For animations of the `ViewPoint` of three-dimensional images one may wish to use the option `SphericalRegion -> True`. This maintains the same scaling as the object is rotated.

6.4 Examples of Graphics Programming

Having seen how to construct functions for built-in commands we can go further and build our own graphics commands. Commands such as `Plot` evaluate their input function over a range and construct a collection of graphics primitives. We could do something very similar. Of course it would not be so useful to reproduce the actual built-in commands. However there are many other tasks to be tackled by writting code in the *Mathematica* programming language. Here we shall look at some possibilities.

■ 6.4.1 Iterative Maps

The first idea comes from iteration. If one repeatedly applies a function, the values returned in some cases converge to a fixed point and in others to seemingly random and complex behavior. For example, consider a function such as $4x(1-x)$.

Floating-point numbers are used to ensure real arithmetic.

```
In[49]:= fun[x_] := 4. x ( 1. -x )
```

This is the value for an argument of .25.

```
In[50]:= fun[ .25]

Out[50]= 0.75
```

This gives me four iterations of f.

```
In[51]:= fun[ fun[ fun[ fun[ .25]]]]

Out[51]= 0.75
```

Of course *Mathematica* gives a much neater way to apply this function repeatedly.

```
In[52]:= Nest[ fun, .25, 10]

Out[52]= 0.75
```

NestList keeps the intermediate values.

```
In[53]:= NestList[ fun, .25, 10]

Out[53]= {0.25, 0.75, 0.75, 0.75, 0.75, 0.75, 0.75, 0.75, 0.75,
  0.75, 0.75}
```

Now we can change the starting point and we see the function does not converge anymore.

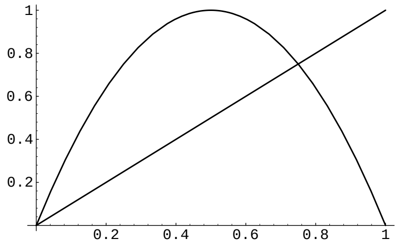
```
In[54]:= NestList[ fun, .24, 10]

Out[54]= {0.24, 0.7296, 0.789135, 0.665603, 0.890303, 0.390655,
  0.952175, 0.182151, 0.595887, 0.963222, 0.1417}
```

Very nice, but I want to see a graphical representation of this. To do this I shall write some code. The function is being applied to some point, and the value which results is used as the next starting point. We are thus interacting with the line $< x, x >$ and we are interested in the two lines $< x, x >$ and $< x, f[x] >$.

These are the two lines we are interested in.

```
In[55]:= plot = Plot[ {x, fun[x]}, {x,0,1}]
```



This function collects all the steps which we have carried out. It takes two arguments and returns a Graphics object.

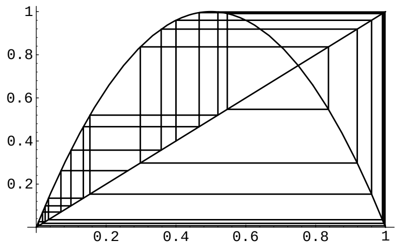
```
In[56]:= Iterate[ st_, n_] :=
           Block[{ res},
             res = NestList[ fun, st, n] ;
             res = Flatten[ Transpose[ {res, res}]] ;
             res = Partition[ res, 2,1] ;
             Graphics[ Line[ res]]
           ]

In[57]:= Iterate[ .4, 20]

Out[57]= -Graphics-
```

This combines the result.

```
In[58]:= Show[ plot, %]
```



In a few lines of code we have a useful way to explore these iterated procedures. This function Iterate could be extended to take the iterated function as an argument and also to generate all the plots automatically.

6.4.2 Correlation Plots

As another example I shall try to build a new type of graphic. *Mathematica* comes with many packages to generate particular types of graphics such as pie charts, bar charts or log plots. We shall look here at another type of plot: one which looks for correlation between sets of data by graphical means.

This is a simple data set.

```
In[59]:= data = Table[ {x, x^2}, {x,5}]

Out[59]= {{1, 1}, {2, 4}, {3, 9}, {4, 16}, {5, 25}}
```

This splits up the list.

```
In[60]:= res =
        Table[ Map[ {Part[ #, i], Part[#, j]}&, data],
          {i, 2}, {j, 2}] ;
```

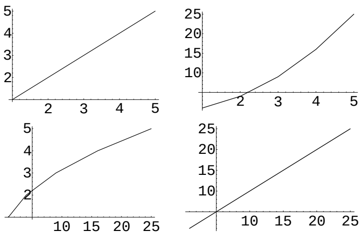
This makes a `GraphicsArray` of the result.

```
In[61]:= res = GraphicsArray[
          Map[ ListPlot[#,
                    PlotJoined -> True,
                    DisplayFunction -> Identity]&,
          res, {2}]]

Out[61]= -GraphicsArray-
```

We can show the result.

```
In[62]:= Show[ res]
```



This is a more complex version. It reverses the data so that the diagonal goes from lower-left to upper-right. It also sets graphics options in the result.

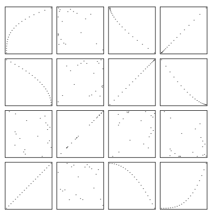
```
In[63]:= CorrelationPlot[ data_] :=
        Block[{ i, j, res, len = Length[ First[ data]]},
          res = Table[ Map[ {Part[ #, i], Part[#, j]}&, data],
              {i, len}, {j, len}] ;
          res = Reverse[ res] ;
          res = Map[ ListPlot[ #, Frame -> True,
              FrameTicks -> None,
              AspectRatio -> 1,
              DisplayFunction -> Identity]&,
              res, {2}] ;
          Show[ GraphicsArray[ res]]
        ]
```

This is a more complex data set. We shall suppress the output.

```
In[64]:= data = Table[ {x, Random[], -x^2, x^3}, {x,0,1,.05}];
```

This is the result. It neatly shows the relationships between the sets of data.

```
In[65]:= CorrelationPlot[ data]
```



■ 6.4.3 A New Primitive

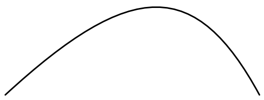
Mathematica graphics provide many primitives for plotting. However there may be some primitive which you desire but which is not present. Instead of saying “if only they had ...” why not actually write it. Here we shall look at an example of building a new primitive to draw a Bézier curve.

The chapter which describes *Mathematica* PostScript shows how the `PostScript` graphics primitive can draw directly in PostScript and the procedures `MBeginOrig` and `MEndOrig` could switch in and out of user coordinates. We can thus use this with the PostScript operator curveto to write a Bézier primitive.

```
In[66]:= Bezier[ {{x0_, y0_}, {x1_, y1_},
                  {x2_, y2_}, {x3_, y3_}}] :=
           {PostScript[
             StringJoin[ ".004 setlinewidth MBeginOrig ",
               ToString[ x0], " ", ToString[ y0], " moveto ",
               ToString[ x1], " ", ToString[ y1]," ",
               ToString[ x2], " ", ToString[ y2]," ",
               ToString[ x3], " ", ToString[ y3], " curveto "]],
            PostScript["MEndOrig stroke"]}
```

The `PlotRange` must be set.

```
In[67]:= Show[ Graphics[ Bezier[ {{0,0}, {2,2}, {3,2}, {4,0}}],
                PlotRange -> {{-1,5},{-1,3}}]]
```



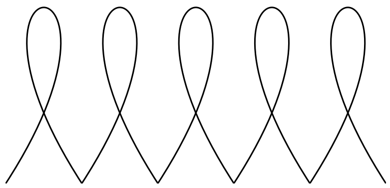
We can make a nicer example of the use of our new primitive.

```
In[68]:= fancy[ {x0_, x1_}] :=
                Block[ {d = x1 - x0},
                        {{x0, 0},
                         {x1 + d, d/2.},
                         {x0 - d, d/2.}, {x1, 0}}
                ]
```

```
In[69]:= data =
           Map[ fancy,
               Partition[ Table[ x, {x, 0,5,1}], 2, 1]];
```

```
In[70]:= data = Map[ Bezier, data];
```

```
In[71]:= Show[ Graphics[ data, PlotRange -> {{0,5},{0,.5}}]]
```



■ 6.4.4 A *MathLink* Renderer

As a final suggestion for programming *Mathematica* graphics one may wish to consider using *MathLink* to write your own renderer. *MathLink* will send the `FullForm` of an arbitary *Mathematica* expression to an external process. For example, you could make a function which will render a `Graphics` object and take advantage of some specialization which *Mathematica* does not normally support.

Then you could make your `$DisplayFunction` call this function and all the other commands such as `Plot` and `Show` will just work as before.